Facultat d'Informática de Barcelona

Universitat Politècnica de Catalunya

Enginyeria Tècnica de Sistemes

Final year project

JSMapper

A kernel-based keyboard & mouse input events generator for game devices on Linux

---

Author:
Eduard Huguet Cuadrench
eduardhc@gmail.com

Director:
Lluis Pérez Vidal
lpv@lsi.upc.edu

March, 2013

Contents

1 Introduction **4**
- 1.1 What is JSMapper . . . 4
- 1.2 Motivations . . . 4
- 1.3 Objective . . . 5
- 1.4 Document Structure . . . 5

2 Preliminary Analysis **6**
- 2.1 Overview . . . 6
- 2.2 The Linux Kernel Input Subsystem . . . 6
  - 2.2.1 Under the hood . . . 7
  - 2.2.2 Hardware drivers . . . 7
  - 2.2.3 Input core . . . 8
  - 2.2.4 Event handlers . . . 9
- 2.3 Joystick support on Linux . . . 10
- 2.4 How JSMapper will work . . . 12

3 Specification **13**
- 3.1 Overview . . . 13
- 3.2 Functional Requisites . . . 13
  - 3.2.1 Use Cases . . . 13
  - 3.2.2 Target Users . . . 14
- 3.3 Non-functional Requisites . . . 15
  - 3.3.1 Usability . . . 15
  - 3.3.2 Stability . . . 16
  - 3.3.3 Maintainability . . . 16
- 3.4 Conceptual Model . . . 17

4 Design **18**
- 4.1 Overview . . . 18
- 4.2 Kernel module . . . 18
  - 4.2.1 Frontend . . . 19
  - 4.2.2 Core . . . 20
  - 4.2.3 Event generator . . . 20

4.2.4 Event filtering . . . 21
4.3 C/C++ library . . . 22

5 Development **23**
5.1 Overview . . . 23
5.2 Project structure . . . 23
5.2.1 Kernel module . . . 24
5.2.2 C/C++ Library . . . 25
5.2.3 Unit tests . . . 26
5.2.4 CLI tools . . . 26
5.2.5 Documentation . . . 26
5.3 Compiling the project . . . 28
5.3.1 Pre-requisites . . . 28
5.3.2 Code checkout . . . 28
5.3.3 Compiling and installing . . . 29
5.4 Tools used . . . 29

6 Analysis **31**
6.1 Mapping . . . 31
6.1.1 Controls . . . 31
6.1.2 Actions . . . 33
6.1.3 Modes . . . 35
6.2 Driver API . . . 37
6.3 Profiles . . . 38
6.3.1 Device Maps . . . 38
6.3.2 Profile loading . . . 39

7 Planning & Economical Analysis **40**
7.1 Planning . . . 40
7.2 Economical analysis . . . 41

8 Conclusions **44**
8.1 Achievements . . . 44
8.2 Future . . . 45
8.2.1 Improvements . . . 45
8.2.2 Kernel inclusion . . . 46

Glossary **47**

A User manual **53**
A.1 Overview . . . 53
A.2 Device Maps . . . 54
A.2.1 Creating a device map . . . 55
A.2.2 Why device map files are needed? . . . 55
A.3 Creating a Profile . . . 57

A.3.1 Game profiles 57
A.3.2 Creating actions 57
A.3.3 Mapping actions 62
A.3.4 Modes 64
A.3.5 Element names 66
A.4 Using profiles 68
A.4.1 Loading a profile 68
A.4.2 Clearing device 68

Chapter 1

Introduction

1.1 What is JSMapper

JSMapper is a Linux joystick mapping tool designed for gamers, who want to take full profit of their game devices under this operating system. It allows the user to generate keystrokes, mouse events, and complex key sequences and map them to any button or axis, while also supporting advanced features such as different mapping modes, which can be selected based on either button states or axis positions. What makes JSMapper special is that it operates at kernel level, so it can be used either for games running on Linux, Windows games running under Wine, or even Windows games running under a virtual machine on a Linux host.

1.2 Motivations

There's a huge number of gaming devices available for personal computers these days. They range from simple joysticks featuring only an stick and a couple of buttons, to advanced, specialized devices such as HOTAS combos (usually splitted in two devices, an stick part and a throtle part), or wheel drives featuring 3-pedals sets (or even a seat!).

These kind of devices are usually shipped along with specific software tools for Windows, so the user can fully program the device for the game to play in any desired way. Although most of the existing game devices are recognized by Linux, no software solution is available for this OS to take full advantage of the advanced features of these devices, such as the ability to define different operating modes (which can be selected using an switch

on the device itself), so different commands can be assigned to the same button. This is a really useful feature for i.e. flight simulation games, where user can define different profiles for navigation, air-to-air and air-to-ground modes, and select between them simply by selecting the appropiate switch position on the device.

The reason to start JSMapper project was to have a similar solution for Linux, which also could be as universal as possible so any gaming device under this OS could provide similar functionality to that offered by their propietary solutions available for Windows.

1.3 Objective

As stated above, the objective for this project will be to create an universal game device mapping system for Linux, so any supported joystick, wheel drive or similar device can be now rendered fully programmable by the user. Also, the resulting system not only will render this feature available to pure Linux games, but also to Windows games running either under Wine or a virtualized Windows machine on Linux.

1.4 Document Structure

The first part of the document (chapters 2 “Preliminary Analysis” and 3 “Specification”) shows a detailed analysis of how the input subsystem works in Linux kernel, plus how it might be improved by means of the capabilities offered by *JSMapper*, regarding the usage of advanced game devices.

Second part is composed by chapters 4 “Design”, 5 “Development” and 6, which offer a comprehensive view about the project, ranging from architecture design to development issues and technical decisions made during the development phase.

Third part (chapters 7 “Planning” and 8 “Conclusions”) contains a reference about how the project development has been structured in time, plus a discussion of possible improvements that could be added in successive development phases.

Finally, fourth part is composed of appendix A, containing the reference manual for final users.

Chapter 2

Preliminary Analysis

2.1 Overview

This chapter presents a preliminary analysis about JSMapper project. As a start, it includes an overview of how Linux kernel input subsystem works, how is the current state of joystick support under Linux, and how JSMapper approach fits into this schema to improve the situation.

2.2 The Linux Kernel Input Subsystem

The input subsystem is the part of the Linux kernel that manages, in a unified way, all the different input devices (such as keyboards, mouses, graphic tablets, joysticks, etc...) that can be attached to the system. It was designed with the clear goal in mind of hiding, almost completely from outside the kernel, the specific hardware interface (USB, PS/2, serial, etc....) the device was using to connect to the computer, while presenting an unique, consistent API interface to userspace that could easily be used by all the other system components that needed to deal with user input, like i.e. the console process or the X11 graphical window system.

The current Linux input subsystem is based mainly on the work of Vojtech Pavlik, who provided the initial implementation for a flexible joystick support API in 2.3 version, while also improving global USB support in the kernel. The work on the subject continued all along 2.4 and 2.5 versions, and finally in 2.6 the new, unified input subsystem fully replaced the old, hardware specifc support that previously existed on the kernel.

2.2.1 Under the hood

The input subsystem is mainly divided in three components, whose relationship is shown in the figure 2.1 below:

- the hardware drivers
- the input core
- the event handlers

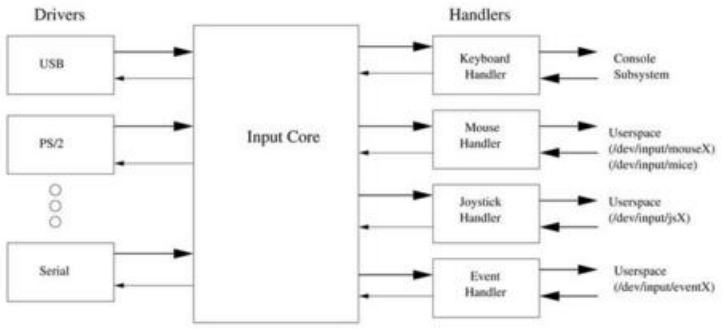

Figure 2.1: The Linux kernel input subsystem architecture

These three elements of the input subsystem communicates each other by using *events*, which are structures defined at kernel level containing all the information associated with a given input event, such as the button being pressed or released, its current state, the timestamp of the action, etc...

Note that, while most communication is done from hardware to drivers, from these to core, then to event filters and finally reaching to userspace, communication can also be done in the opposite way, this is, from userspace to hardware devices: this is how is possible i.e. to manipulate keyboard LED status, or to send i.e. motion commands to force-feedback capable driving wheels.

2.2.2 Hardware drivers

Hardware drivers are the part of the Linux input subsystem that directly interacts with the hardware. This is, there are a large number of available hardware device drivers. Most of them lay under the *drivers/input* folder and subfolders of kernel source tree. There, they are organized either by bus type used to interact with the computer (*gameport*, *serio*, ...), or else by their device class type (such as *joystick*, *mouse*, *keyboard*, ...).

Some other drivers lay into dedicated subtrees under Linux kernel. Among them, one of ther most important driver today for the input subsystem is the generic *HID* driver (*HID: Human Interface Device*), which implements support for the "USB-HID" specification. Effectively, most of the USB-based input devices available these days, including joysticks and similar, declare themselves as HID-compliant devices: this is a standard that provides an unified access model to the different elements conforming the device, such as buttons, keys, relative and absolute axes.

By fully supporting HID-class specification, the number of supported input devices in Linux has raised dramatically, with dedicated hardware drivers being needed only for very specific tweaks.

In any case, the main function of any input subsystem hardware device driver is always the same: to deal with the specifics of the device and/or bus type, then generate an unified set of *events* that will get injected into the input core, and later received by the event handlers attached to it.

2.2.3 Input core

The input core is the central part of the input subsystem, as it provides the needed communication between all the different components that conforms the input mechanism.

Among the different tasks assigned to the input core, the main one of them is to receive the event notifications provided by the low-level hardware drivers, then route them to the different event handlers attached to the input core. The event handlers then can take the appropiate actions, which usually consist in some kind of notification to userspace.

Another one of the tasks performed by the input core is the responsability of notificating event handlers about new devices being plugged in (or removed from) the system, so event handlers can then decide if they must listen to the events provided by that particular device, or simply ignoring it.

For instance, the existing *joydev* module attaches to any joystick-type device attached, by querying if the device supports the specific subset of axes and key (button) ranges assigned to joystick devices. If so, then it creates a kernel device node (*/dev/input/js0*, i.e.) that provides userspace with joystick API for that particular device.

2.2.4 Event handlers

Event handlers are basically at the receiving end of the kernel input subsystem architecture, and the final responsible of providing the userspace API for all other layers on the OS. Their role is to receive and dispatch the event notifications sent by the low-level hardware drivers of their interest, then provide an uniforn, consistent API to userspace for the device type they represent.

Effectively, every device handler provides a different API, which is intended to be useful for the particular type of device it represents. To do so, they register themselves as a "handler" inside the Linux kernel, which causes input core to notify them about any new (or existing) input device attached to the system: then, the event handler decides if such a device if of its interest (by inspecting some device flags) and, if so, creates a new device node (under */dev/input/*, usually, which provides the userspace API for that device.

These are some of input event handlers currently available in Linux kernel:

- *evdev*: generic input event handler. This is the primary input interface used by current Linux software layers (including X11 server, terminal emulators, etc..., and it's primarily used to get input events from both mouse and keyboards attached to the system.
- *mousedev*: old mouse interface, mostly used to provide compatibility with old PS/2 mouse interface.
- *keybdev*: also an old interface, this time used to provide compatibility with old VT keyboard API
- *joydev*: joystick interface, providing existing Linux joystick API.

The way is designed, devices and event handlers are completely detached each other: this means, an event handler will simply provide a consistent input API to any device matching an specific set of characteristics, no matter how the device is actually built ot connected to the system. Also, the same device will sport more than one event handler, if it matches the characteristics required by more than one of them: for instance, a USB mouse / keyboard combo will feature two *event* nodes (provided by the generic *evdev* event handler), a *mouse* node (provided for compatibility by the *mousedev* handler), and maybe a *kbd* node if the old *keybdev* handler is loaded.

Event filtering

A very interesting feature of the event handlers (and of particular interest for JSMapper) is the possibility to register themselves as a *"filter"* for the device, instead of as a regular handler: if so, then the input core will send any event first to *filter*-type handlers, which has the possibility to filter out the event, and only after all filter-type handlers have dispatched the event (and only if none of them has blocked it), the event will reach the regular handler.

This is the method used by JSMapper to early intercept the events sent by the joystick device, and convert them to keyboard and mouse events without letting *joydev* event handler to dispatch them.

2.3 Joystick support on Linux

As yet stated, current Linux support for joysticks is implemented by means of an special input event handler (*joydev*), which detects an provides a joystick-type API for any device plugged into the system featuring any of the following items:

- An absolute axis identified either as "X", "Wheel", or "Throttle"
- Any buttons whose IDs lay into into the range assigned for joysticks and gamepads

For such devices, the *joydev* event handler will create a node (usually named */dev/input/js0* for the first device,*/dev/input/js1* for the second and so on...), which userspace programs will use to access the joystick and take profit of it for games and so.

A typical sequence for a program using joystick support is to open the *js* device created by *joydev*, then using blocking *read()* calls on it to read the *"js_event"* structures posted by the driver indicating the different events occurred. Listing 2.1 shows the typical workflow of a program using joystick API on Linux:

Listing 2.1: Typical program flow for joystick Linux API

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/joystick.h>

#define JOY_DEV "/dev/js0"
```

```
int main()
{
    struct js_event js;
    int joy_fd;

    joy_fd = open( JOY_DEV , O_RDONLY);
    if( joy_fd == -1 )
    {
        printf( "Couldn't open joystick\n" );
        return -1;
    }

    while( 1 )  /* infinite loop */
    {
        /* read next joystick event */
        read(joy_fd, &js, sizeof(struct js_event));
        /* do domething useful with it... */
    }

    close( joy_fd );
    return 0;
}
```

Alternatively, programs can use a more sophisticated approach by using *select()* calls to detect when there is data (events) available for reading, or simply using the file handle in non-blocking mode, which will make the *read()* call to return -1 in case there is no data available for reading.

In addition to the events returned by the *read()* call, the API also features the following IOCTL control codes:

- **JSIOCGVERSION**: returns API version
- **JSIOCGNAME**: returns device identifier string
- **JSIOCGBUTTONS**: returns number of buttons on the device
- **JSIOCGAXES**: returns number of axes on the device
- **JSIOCGCORR**, **JSIOCSCORR**: gets / sets axis correction values
- **JSIOCGAXMAP**, **JSIOCSAXMAP**: gets / sets axis mapping
- **JSIOCGBTNMAP**, **JSIOCSBTNMAP**: gets / sets button mapping

Listing 2.2 shows an example of how IOCTL calls can be used to retrieve some device attributes:

Listing 2.2: Using joystick IOCTL calls

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/joystick.h>

#define JOY_DEV "/dev/js0"

int main()
{
    struct js_event js;
```

```
    int joy_fd, num_of_axis=0, num_of_buttons=0;
    char name_of_joystick[80];

    joy_fd = open( JOY_DEV , O_RDONLY);
    if( joy_fd == -1 )
    {
        printf( "Couldn't open joystick\n" );
        return -1;
    }

    ioctl( joy_fd, JSIOCGAXES, &num_of_axis );
    ioctl( joy_fd, JSIOCGBUTTONS, &num_of_buttons );
    ioctl( joy_fd, JSIOCGNAME(80), &name_of_joystick );
    printf("Joystick detected: %s\n\t%d axis\n\t%d buttons\n\n"
        , name_of_joystick
        , num_of_axis
        , num_of_buttons );

    close( joy_fd );
    return 0;
}
```

2.4 How JSMapper will work

Based on this preliminary analysis, JSMapper will use the services provided by the input subsystem to achieve the desired functionality, which is the ability to intercept josytick events at a very low-level, then map then to simulated keyboard and mouse events, which will be programatically defined through a userspace API.

More specifically, JSMapper will be built as a kernel module (*jsmap-perdev*), which will behave in the following way:

- it will declare itself as *event handler* for joystick-alike devices, just as existing *joydev* handler does, so it will get notified every time a device of such a type is plugged into the system.
- it will register explictly as a *filter* for those devices, which will allow it to receive and intercept events **before** they reach the *joydev* handler.
- it will use a *virtual event generator* to generate and inject "fake" mouse and keyboard events into the input core, which will get routed in the usual upstream to userspace.

Also, for every joystick device attached to the system it will create a device node (*/dev/input/jsmap0* for the first device, etc...), that will be the entry point for the programming API offered by the module. This API will allow external programs, such as the provided CLI tools, to program the device to map mouse and keyboard actions to the device elements, such as button and axes.

Chapter 3

Specification

3.1 Overview

This chapter presents the functional and non-functional requisites of the project, including usability constraints and possible use-cases.

3.2 Functional Requisites

3.2.1 Use Cases

The main use case for this software will be a user wanting to "program" its game device, either a joystick, a wheel drive or a gamepad i.e., to generate keystrokes and mouse movements that cause actions on the game being played when operating on the buttons. The range of actions that can be simulated varies from issuing simple keystrokes, mouse movements and clicks, to complete keyboard macro sequences, which would allow him or her to i.e. display a menu using a keyboard shortcut, moving through the available options in it by using the cursor keys, then selecting the desired option by issuing the ENTER key. A whole sequence such as the one described cold be mapped to a button on the device, so the user can trigger it simply by operating the button.

As the implementation allows for advanced features such as modes and shift state buttons, the user will also be able to simultaneoulsly load different mapping schemes that are appropiate for differents phases of the game, and then switch between them by toggling a button on the device or by moving an axis between certain positions.

The user is expected to create "profiles" for the games to play, each of one containing the mapping schema to apply (including the possible modes, etc...) for the game, and then load them into the driver before starting the game.

A secondary use case could be to use the game device to control not a game, but either the desktop itself or an specific software like a word processor or similar. Although not the intended use case (the software was created having games in mind), it could also be perfectly used for such a task.

3.2.2 Target Users

The project is initially intended for Linux users who want to take a better profit of their game devices under this OS, by providing "programming" capabilities to their device in a way similar to what is offered for advanced devices on the Windows platform.

Also, the target user is anyone wanting to run "games" under the Linux operating system, be it native Linux games or Windows games running under Wine or a virtualized Windows session on a Linux host. The software is specially suitable for "complex" games such as flight simulators, as they usually feature a very large set of commands that can be invoked using shortcuts and that can bebefit from the ability to map such actions to device controls.

Although no special abilities are required to use the software (compiling and installing it's a whole different story, and in a future it should be made by Linux distribution packagers), the user should at least know how to edit an XML file using an standard text editor such as *Kate*, *GEdit*, etc..., and how to invoke executables from the shell command line interface (CLI), as this will be the preliminary way to interact with the kernel module, at least until a GUI frontend is developed. However, this capabilities should be taken for granted for any minimally experienced Linux user.

3.3 Non-functional Requisites

3.3.1 Usability

General Usage

Main usability constraint is to keep the required competences for the common target user as low as possible. With this in mind, the only required abilities on to use the software are the following:

- Editing text files (XML) using a text editor
- Using command line

As mentioned before, this capabilities should be taked for granted for any minimally experienced Linux user. The only drawback could be the fact that profile files are in XML format, which can be a little confusing to those not familiarised with it.

However, a GUI frontend is planned which would allow to create game profiles in a graphical way, so this will contribute to set the software usability some degrees higher.

Installation

Setting up the project and compiling is a little more complicated, as it involves some more skills that simply using it. As this task is intended to be performed by Linux distruibution packagers, it's analyzed separately.

Anyway, a user wanting to compile and install the software directly from the source code should be competent in the following areas:

- Be able to else checkout the code from a remote Git repository, or to download and extract a TAR file containing an snapshot.
- Be able to install the required dependencies (as mentioned in "Development" chapter) needed to compile the project.
- Be familiarised with compiling and installing kernel modules, including necessary steps such as installing the appropiate kernel sources and prepare them to compile external modules.
- Know how to compile a *cmake*-based project.

See section 5.3 on chapter 5 for detailed instructions about how to compile and install the project from the source code.

3.3.2 Stability

The stability of the code base is must, specially considering that the crucial part of the software is a kernel module running in privileged kernel space. This means that, if the module behaves badly then it can affect the integrity and stability of the whole system, and not only that of a single program.

For such a reason, special care is taken when passing parameters between user space and kernel space, in order to avoid accidental (or intended) buffer overflows, out of bounds accesses, etc... The kernel module code also tries to ensure always that concurrent access to common variables is properly regulated using mutexes and spinlocks. This is specially important as notifications from the input system core usually gets called from within an IRQ context, so they can be received at any time (even during profile loading sequence).

3.3.3 Maintainability

Maintability of the code is granted, given it's not tight to any version-specific feature of any of the dependencies required:

- The current Linux kernel input subsystem was designed around 2.4 version, and hasn't suffered major changes since that nor it's expected to suffer major changes in future revision. The project itself has been succesfully compiled against kernel versions ranging from 2.30 to 3.6.9, the current stable version at time of writing.
- The C/C++ library is based mainly on GNU's standard *libstdc++* library, which is a very mature and stable product. Other dependencies used, such as *libxml2* and *libncurses*, are also in a full mature state and shouldn't suffer major changes in the future.

With these constraints in mind, maintanability of the project should not pose any major problems nor in the short nor medium terms envisaged.

3.4 Conceptual Model

The conceptual model of the software is more extensively described in chapter 4, "Design". However, it can be quickly summarized by dividing the project structure in 3 main components:

- A kernel module which acts an *event filter* for the game devices, and which intercepts events from the hardware device driver and converts them to simulated mouse and keyboard actions.
- A C/C++ library which wraps access to the kernel module, by providing easy-to-use classes that deal with the IOCTL API intrinsics.
- A set of *CLI userspace tools*, which are what the user will use to interact with the driver.

Chapter 4

Design

4.1 Overview

This chapter provides a description of the whole JSMapper project architecture design, and the relationship between the different components that take part of it.

As seen at the end of chapter 2, JSMapper main component is a kernel module implementing an special event handler for joystick-type devices. Additionally, it includes a couple more components:

- A C++ library, which wraps access to the underlying kernel module through a set of useful classes.
- Some userspace tools, which are used to interact with the kernel module from the command line.

The following sections provide an in-depth explanation of the components above.

4.2 Kernel module

As yet mentioned, the core of JSMapper is implemented using a kernel module, named *jsmapperdev*. This core is compiled against the kernel sources, then installed along with the other kernel modules so it can be automatically loaded by the system.

The module in itself is internally divided in the 3 components:

- *Frontend*: it handles the whole module initialization and registration into the input core. It handles also the kernel notifications such as target device connection & disconnection, and the processing of event filtering function and API IOCTL calls.
- *Core*: it handles the data structures needed to map device buttons and axes to mouse and keyboard actions.
- *Event generator*: it implements the virtual event generator, which is used to inject the "fake" into the input core.

Figure 4.1 illustrates the relationship between the above components:

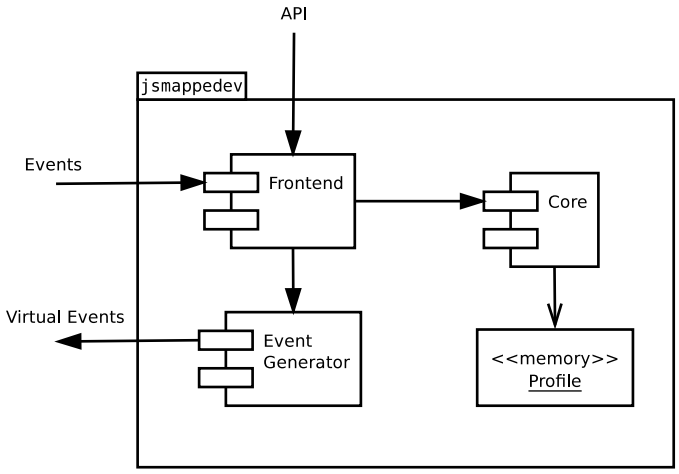

Figure 4.1: Kernel module components

4.2.1 Frontend

The module frontend contains most of the *glue* code needed to properly implement a Linux kernel driver. This is, it's the responsible of actually registering the module, receiving notifications on devices being plugged and/or unplugged, etc...

More specifically, the module frontent performs the following tasks:

- *Module initialization*: registering the module as an event handler filter for joystick type devices, in the same way *joydev* module does. The virtual event generator is also created during module initialization.

- *Device detection*: device connection and disconnection notifications are also handled by the frontend. For new devices being plugged in, it creates an associated *jsmap* node for it, which will be used to *program* the device using the API. For devices being removed, it simply destroys and cleanup the previously existant associated device node.
- *Handling API calls*: the API IOCTL calls issued to the *jsmap* nodes are also received by the module frontend. Its usual role here is to copy and decode the parameters received from userspace into kernel space, then forward the call to the module *core*. In some cases, it will also reencode and copy back to userspace the data to return, if any.
- *Event filtering*: events sent by the associated hardware device driver are also received by the frontend, who performs the filtering and mapping into actions (more on this below).
- *Cleanup*: finally, cleaning up of all the associated resources at module unloading is also performed by the this component. This includes freeing all associated structures, device nodes, destroying the event generator, and finally unregistering the module from the chain of input event handlers.

4.2.2 Core

The core contains all the needed structures to keep track of the association between device elements (buttons, axes) and target actions.

On device connection, the frontend creates an initializes an instance of the "core" structure (a *jsmapdev_core* struct, specifically) for that device, which then gets associated to the *jsmap* device node also created for the attached device.

The "core" structure for a device can be then modified by means of API IOCTL calls made to the matching *jsmap* device.

4.2.3 Event generator

Also during module initialization, a *virtual event generator* is created: this is done by registering a new input "device" into input core, just as any true hardware device driver does. Only difference is that in this case the device is not attached to any "real" bus, but to a "virtual" one.

Other than that, the event generator announces itself as capable of producing keyboard and mouse events, so the system reacts to it just as it

would react if a real keyboard and mouse would have been attached to the system: by creating appropiate input event handlers that will render the device visible to userspace (i.e., X11 server will start processing inputs from it without any further intervention, thanks to HAL & D-BUS).

4.2.4 Event filtering

Event filtering is performed collaboratively between the three sub-components of the kernel module:

- Module frontend receives events through the *filter* callback function declared when registering the event handler.
- The filter function checks core to see if the source button (or axis) has any action mapped to it.
- If so, then it launches the simulated action through the event generator, then discards the original event thus avoiding *joydev* handler processing it.

Figure 4.2 displays the path followed by an event, since its inception on the hardware device, the filtering through *jsmapperdev*, and its final destination into *joydev* (the standard input handler for joystick devices), which in its turn forwards it to userspace through its own API:

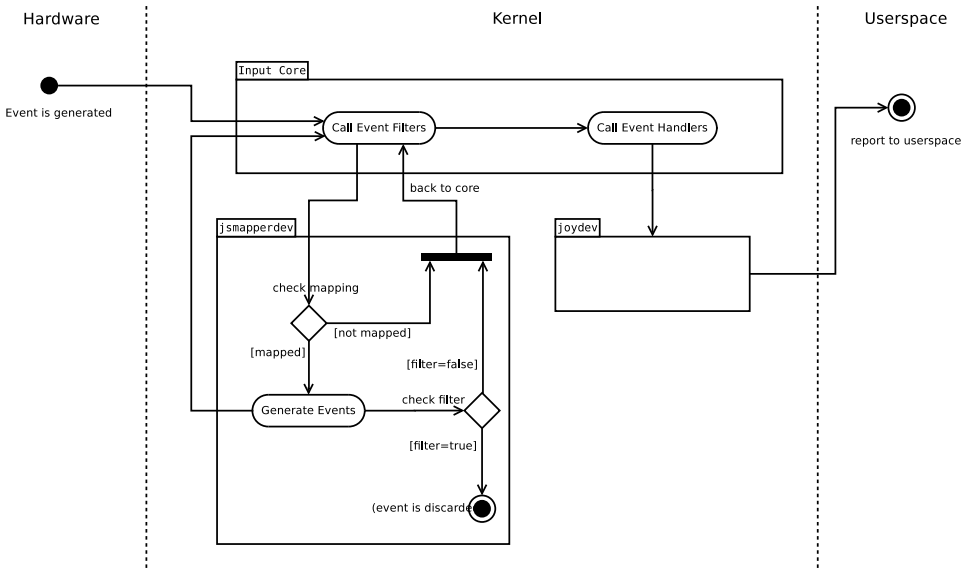

Figure 4.2: Event filtering flowchart

As seen on the figure, generated events get also injected into input core in the exact same way that "real" events do. The *jsmapperdev* module recognizes its own "fake" events and let them pass without any filtering, to avoid recursive calls.

4.3 C/C++ library

In order to make access to the module API easier from a developer's point of view, the project features also a C++ class library that encapsulates the access to the IOCTL API calls, using easy-to-use classes to build up profiles programatically profiles and send them to the driver.

In addition, the library provides also useful methods to save & restore the profiles as XML files.

Here's a summary of the classes provided by the library:

- *Device* class: wraps access to *jsmap* device nodes, providing services to query associated device status and attributes, and setting up action mapping.
- *Action* class and subclasses: provide an easy way to create the actions to be associated to device elements.
- *Mode* class: represents a set o assignments between device elements (buttons, axes) and actions to launch for them. A *Mode* object might contain an arbitrary number of child submodes and, unless it's *root* mode, an activation *Condition* object.
- *Profile* class: glues altogether by contaning the *root* mode (plus their submodes, if any), and provides the XML serialization services needed to load & save profiles from &to disk files.

All classes are contained into *jsmapper* namespace, for name collision avoidance.

Chapter 5

Development

5.1 Overview

This chapter presents some notes regarding project development, along with a description of how code is structured, tools and libraries used, and compiling pre-requisites and procedures.

5.2 Project structure

The whole project has been structured as a *cmake*-based project, as this popular open source, cross-platform compiling tool provides the necessary flexbility to joint the different parts of the project. Thus, a single, top-level *CMakeLists.txt* file (the *cmake* project file) takes care of building all the components needed to build ther project. This top-level project file includes the nested *CMakeFiles.txt* subproject files belonging to the kernel module, the C/C++ library, the unit tests, and the CLI userspace tools.

Put short, the project is internally structured in 3 parts:

- The kernel module
- The C/C++ library (including unit tests)
- The CLI tools

5.2.1 Kernel module

Kernel module (*jsmapperdev*) code is located under *src/linux/drivers/input* folder, following a layout that mimics that of Linux kernel source.

As *cmake* doesn't support (as of today) a native way to compile kernel modules, an special set of custom commands are used inside the *CMakeFiles.txt* project file for the module. These commands end up by calling the native Linux build system (*Kbuild*) for kernel modules, which relies on scripts located under */lib/modules/<kernel version>/build/* folder.

The kernel module is implemented by this set of source files:

- **jsmapper_api.h**: contains API definitions, such as IOCTL codes and parameter structures. This file is the only one designed to be included both from kernel project and from userspace tools.
- **jsmapper_main.c**: contains the module frontend, consisting in the basic code needed to implement a kernel module, the input event filter callbacks, etc...
- **jsmapper_core.c**: contains the mapping core, this is, the data structures containing the mapping data which get associated to every *jsmap* device node.
- **jsmapper_evgen.c**: contains the code for the virtual event generator
- **Kbuild**: contains the instructions for the Linux kernel build system (*Kbuild*) needed to build the module.

On install, the kernel module will be copied under */lib/modules/<kernel version>/kernel/drivers/input/*.

Event filtering

Event filtering on the target hardware device is provided by the *jsmapper_filter()* function in *jsmapper_main.c* module.

This function is called by the kernel everytime an event is generated from the device: the function then checks if the source event is mapped to any action, and takes the appropiate steps to launch it. Depending on the source element type, the steps performed will be different:

- For buttons, it will *start* the action when the button is being pressed, and *stop* it if the button is being reelased.

- For axes, it will *start* the action when the axis is moved into the band the action is mapped to, and it will *stop* the action when the axis is moved outside of the band.

Finally, it returns a boolean value to the kernel indicating if the source event must be filtered out, or else it must be let go its way to the other attached event handlers.

Event generation

Simulated events get injected into the input core by calling the kernel-provided function *input_event()*, which is the function called also from the "real" device drivers when they notify changes in device controls upon user interaction.

For simple keystrokes and mouse button clicks, the referred function is called directly from the context of the event filtering function mentioned above.

For complex actions, different mechanisms are used to avoid blocking the filtering thread which is being invoked from an IRQ context, as it provides some restrictions: for instance, from IRQ code the *wait()* call (used to wait some milliseconds) can't be used, as the scheduler is not accessible from that point and thus no other thread can be scheduled.

- Macros are sent using a *workqueue* item: this is a low-level mechanism provided by Linux kernel which allows to "queue" some action to be done into a workqueue, serviced by a background thread that gets activated when some job to be done is posted. The service thread doesn't run into IRQ context but normal process context, so it can safely use *wait()* calls to keep the cadence, etc...
- Mouse moving actions are implemented using a *kernel thread*, which keeps sending axis movement events in the background for as long as the action is kept "active. As "workqueue" service thread, this thread runs in normal "process" context, so it doesn't have the restrictions of IRQ context.

5.2.2 C/C++ Library

The C/C++ library (named *jsmapper*) is located under *src/lib/jsmapper* folder. It provides C++ wrapper classes useful to programatically interact

with the driver, without the need of dealing with IOCTL calls. It also provides higher level functions, such as profile XML serialization.

All library classes are located inside *jsmapper* namespace, to avoid name collisions. On install, the library files will get installed under *$PREFIX/lib/* folder.

Figure 5.1 displays the classes provided by the library, along with their inheritance and relationship diagram.

As a side note, the CLI tools are built using the library instead of the IOCTL API directly.

5.2.3 Unit tests

A number of unit tests are included along the library, to test the correct behaviour of the classes provided. These tests are located under *src/test/* subfolder, and they are built around *gtest*, this is, the "Google C++ Test Framework", whose code is also embedded with the project.

Unit tests can be invoked by issuing *make test* command on the build directory (see section 5.3 below). When installing, the executable test binaries will get copied to *$PREFIX/bin/*, along with CLI tools.

5.2.4 CLI tools

The CLI tools are located under *src/bin/* subfolder. Two executable tools are provided:

- **jsmapper-ctrl**: basic CLI tool used to load profiles into the driver, clearing the state, etc...
- **jsmapper-device**: helper tool used to write "device maps" for the attached device

When installing, the executable binaries will get copied to *$PREFIX/bin/* folder.

5.2.5 Documentation

The project code is documented using standard *Doxygen* tags. This applies to both the kernel driver code, the C/C++ library and the CLI tools. A

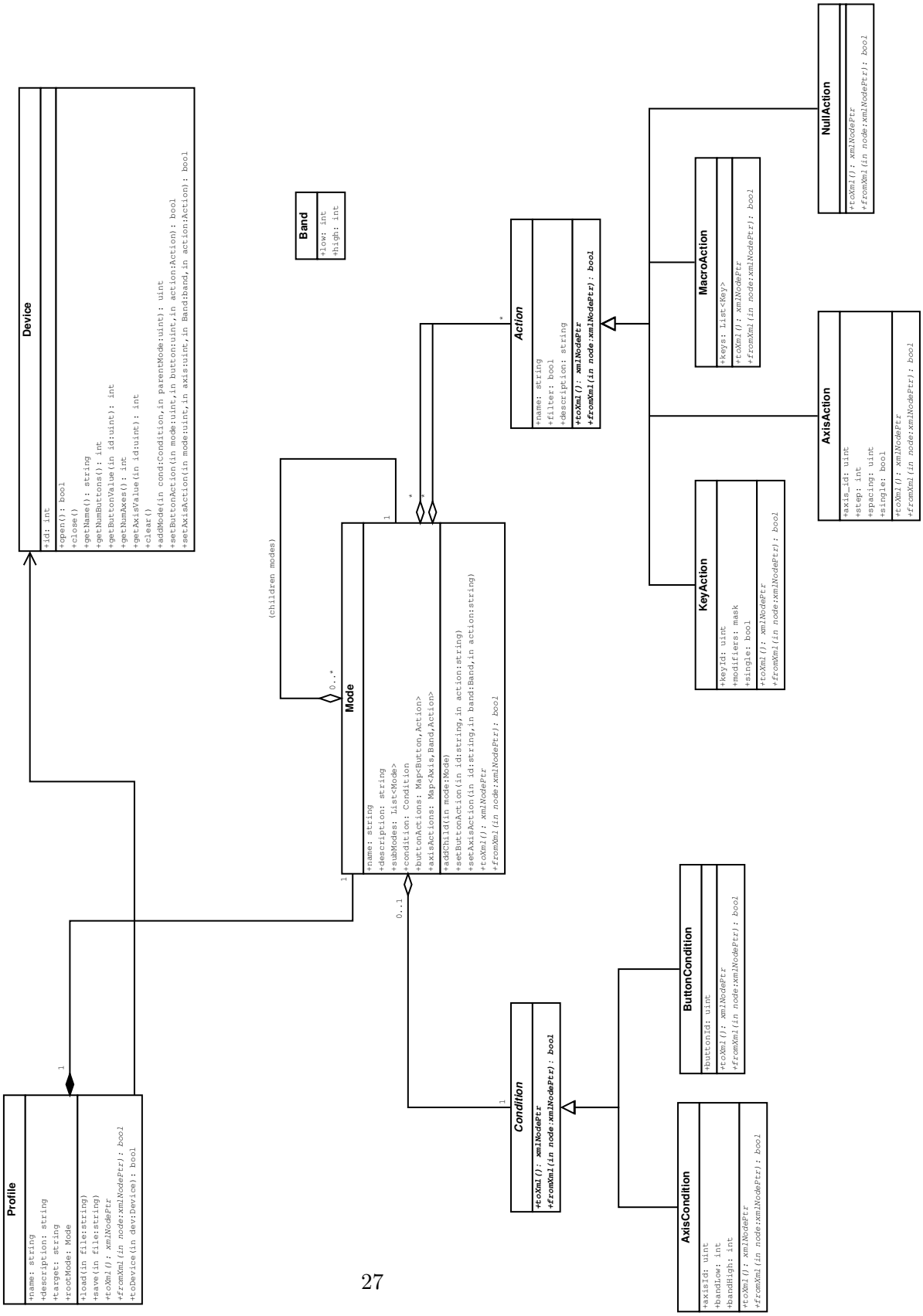

Figure 5.1: Class Diagram

Doxygen configuration file is provided (*doxy.cfg* inside "doc" folder) that can be used to generate the HTML documentation in the following way:

Listing 5.1: Generating HTML documentation

```
$ doxygen doxy.cfg
```

The HTML documentation will be generated under the "html" subfolder.

5.3 Compiling the project

5.3.1 Pre-requisites

The following components must be installed in the system before attempting to compile the project:

- *Git* (for code checkout)
- *C/C++ compiler* (gcc, i.e.)
- *cmake* (the whole project is cmake-based)
- *kernel headers* (to build the kernel module)
- *libncurses5* (development package)
- *libxml2* (development package)

5.3.2 Code checkout

JSMapper code is currently hosted in Assembla, and can be accessed by cloning the remote Git repository into the compiling machine:

Listing 5.2: Checking out the code

```
$ git clone https://git.assembla.com/jsmapper.git
```

This will clone the project into a subdirectory named "jsmapper". The code itself is inside the "src" subfolder.

5.3.3 Compiling and installing

As with any CMake-based project, out-of-the-tree compiling is the preferred way to proceed, so a "build" directory must be created first. Then cmake must be invoked from within the build directory to create the makefiles, and then finally we can go with the usual *"make && (sudo) make install"* sequence.

From the shell:

Listing 5.3: Compiling the project

```
$ mkdir build
$ cd build
$ cmake <path-to-jsmapper>/src
$ make
$ sudo make install
$ sudo depmod -a
$ sudo ldconfig
$ sudo modprobe jsmapperdev
```

This will compile the project and install the binaries into the appropiate system locations. The (*ldconfig*) step might be necessary depending on the system, in order for the installed binaries to find the library.

If no problems are found, then the last step should have loaded the kernel module succesfully, which can be verified by calling *lsmod* and checking that *jsmapperdev* is listed. In case of error, *dmesg* it's usually a valuable source of information.

5.4 Tools used

The following software tools has been used for the project development:

- *QtCreator*, as the main IDE tools
- *GNU C/C++ compiler*
- *openSUSE*, *Fedora* and *Ubuntu* OS, running on the various development and testing PC computers
- *LaTeX* and *Kile* (KDE frontend for LaTeX) for this report
- *GanttProject* for the Gantt diagram and task list
- *Assembla*, as the hosting platform

In addition to the computers used for development, the following two game devices has been used for testing:

- *Saitek X-45 HOTAS*, a very popular device among flight simulator enthusiasts.
- *Logitech Wingman Extreme*, an entry-level joystick
- *Trust FF380 Force Feedback Racemaster*, a wheel drive and pedals combo used for car racing simulators.

The JSMapper features has been tested against the following games:

- *FlightGear*, a very popular open source flight simulator software, running natively on Linux
- *Trackmania*, a Windows car racing simulator running under Wine on Linux.

Chapter 6

Analysis

6.1 Mapping

Mapping *actions* to joystick buttons and axes (*controls* from now on) is the main feature of JSMapper. Its purpose is to be able to map actions such as pressing keys, moving mouse cursor, or even complex key sequences to the controls on any joystick and similar device. When the user operates the device, the specified action is played, and it's interpreted by the system just like if it were coming from a real hardware device.

Concepts related to the *Mapping* feature are discussed below.

6.1.1 Controls

A device *control* is any "mechanic" element on the input device (a joystick, a wheel drive, ...) to be operated by the user. JSMapper is currently able to map actions to the following controls on a device:

- *Buttons*: any kind of simple button, "hat"-type buttons, mode switches, etc..., as long as they are reported as a *button* by the device.
- *Axes*: absolute axes such as X & Y on a joystick stick, throttle, accel / break pedals, etc... Some devices also report "hat"-type buttons as axes instead of buttons.

Any controls is defined as to be able to have always two possible states, "active" and inactive", the exact meaning of which is different for every

control type (see below). This abstraction, though, is used by the mapping engine so any action can be mapped to any control, no matter what their types are.

Buttons

As joystick buttons has only two possible states, pressed and released, theses are obviously mapped to activation states in a very straighforward way:

- Pressed → Active
- Released → Inactive

This means that, when a button is pressed the assigned action will be "active" in its turn, and when the button is released the action will be inactive".

For the case of a simple *Key* action, this means pressing down the mapped key when the button is pressed, and releasing it when the button is released.

Axes

Axis are more complex to map, as they don't provide two activation states but a large, continuous, range of values. For this reason, mapping actions to axes is made by defining *Bands* on it, this is, intervals of values from within the whole range of the axis.

Actions are then mapped to individual *Bands* on the axis, rather that on the whole axis itself. In conclusion, the "activation" state for the band is then defined by the current position of the axis at any time:

- If inside the band → Active
- If outside the band → Inactive

This means that any number of different actions can be mapped to the same axis, as long as different, not overlapping bands are defined for it. Figure 6.1 shows an example of how four actions can be mapped to an axis by assigning them to different bands across the whole axis range, while keeping an "inactive" section at the center (acting as a "dead" zone, i.e.).

The same idea is applied in order to use *axis bands* as activation conditions for *Modes*.

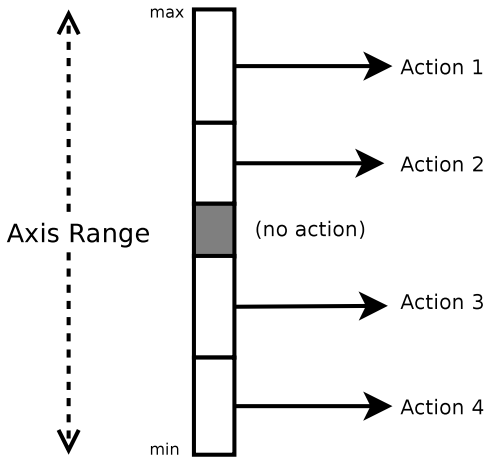

Figure 6.1: Axis mapping through bands

6.1.2 Actions

An *Action* represents the event to "simulated" or ("played") whenever the source control the action is mapped to is operated.

To match *controls* design, actions also have two possible states, "active" and "inactive", the exact meaning of which depends on the action type. The mapping engine will thus "activate" an action when its source control gets "active", and "deactivate" it when control gets back to "inactive" status.

Currently, JSMapper supports 3 types of actions:

- Key actions
- Axis actions
- Macros actions

All action types share a common attribute, named *filter*: if this attribute is set to true, then the original event sent by the hardware input device driver is filtered out, so it's not receuved by the system. If false, then event wil keep its way upstream to the chain of event handlers after executing the associated action.

Key actions

This is the most simple action, designed to simulate key press & release events ("keystrokes"). This action type is suitable to be used for simulat-

ing the entry of either single keys, or either shortcut-alike keystrokes, as it supports an optional "modifiers" mask for this sake.

Mouse button actions are also simulated through this action: in such a case, instead of a key identifier a mouse button identifier is used.

These are the attributes of a *Key* action:

- *key*: Linux kernel code of the key (or mouse button) to simulate.
- *modifiers*: a mask indicating an optional use of *Shift*, *Control*, *Alt* modifiers keys.
- *single* flag: a flag indicating if the key will self-repeat or not

Key actions behaves differently depending on the value of the *single* flag:

- If true, then the key will be "pressed" and immediately released whenever source control gets activated, thus only a single key (or shortcut) will be issued. No action will be taken on source control deactivation.
- If false, then the key (and the modifiers) is "pressed" when the source control gets "activated", and "released" when the source control gets "inactive". As the simulated key will remain "pressed" until then, the system will self-repeat the key for as long as it's hold down, just like if a real key on the keyboard is pressed and held down.

Axis actions

Axis actions are designed to simulate movement in any of the *relative* axes supported by Linux input subsystem, particularly the X & Y axes used by mouse devices. This way, it's possible to use the joystick to move the mouse on screen.

These are the attributes of an *Axis* action:

- *axis*: Linux kernel code of the axis to simulate movement for.
- *step*: an integer values indicating the number of axis units to perform per step
- *single* flag: a flag indicating a single-step movement or a continuous one.

- *spacing*: in continuous mode, the spacing (in ms) between steps

In a similar way to *Key* actions, *axis* also behave differently depending on the value of "single" flag:

- If true, a single movement will be made, of the magnitude indicated by the "step" value. The movement will be peformed at source control "activation".
- If false, then the action will generate a continuous movement on the axis, for as long as the source control is kept "activated".

Macro actions

Macro actions permits automating a number of basic keystroke events which will get executed in a row, thus the name of "macro" assigned to them. Every keystroke of the macro can be defined either as a single key, or either include an optional "modifiers" mask, just like *Key* action.

These are the attributes of a *Macro* action:

- *keys*: an array of keystroke definitions, this is, a key code and an optional "modifiers" mask.
- *spacing*: the spacing left between keystrokes (in ms).

Behaviour of *Macro* actions is always the same: the entire sequence of keys is launched whenever the source control gets "activated", without any possibility to cancel it. No action is done on source control deactivation.

6.1.3 Modes

Modes are an integral part of JSMapper, and one of its most advanced features. Put short, modes are different sets of mapping assignments that coexists at the same time, and which are selectable at runtime by the user by operating i.e. a switch or a button on the joystick.

Modes are organized hierarchically, so there is always a *root* mode, which contains the common assignments (or any at all...), then any a number of optional child submodes (which, in its turn, can contain more submodes by themselves).

Conditions

Children submodes must feature an *activation condition*, which is the condition that must be met in order for the mode to be *active*: this condition can be either a button being pressed, or else an axis fitting a determined range of values (a *"band"*).

Two types of activation condition are supported:

- *By button*: if the button is pressed, the mode is active, else is inactive.
- *By axis band*: if axis current value is inside the band, the mode is active, else is inactive.

Mode switching

At runtime, the mode to be applied for every button (or axis) at a given time is determined in the following way:

- First, starting with root mode, the driver checks if the activating condition of any of its children modes is met. If so, then it repeats the process by starting from that mode, and then recursively, until it can't go further down the mode hierarchy.
- Then, starting from the last mode found, it checks if it contains an assignment for the button and, if so, the associated action is launched. Else, the driver goes up one level and repeats the check with the parent mode, until eventually it gets back to *root* mode if no action is found in any submode.

The way mode selection works, it means that a given child mode can be active *only* if its parent mode is: this is, it's OK to use the samer activation condition for two children modes, as long their respective parent (which must not be the same) have different activation conditions on their own:

Figure 6.2 displays an example of a profile for the Saitek X-45 containing the mandatory *root* mode mapping some basic buttons, plus 3 submodes which get selected by a 3-position switch on the joystick (which internally gets translated to 3 different buttons, labeled "M1", "M2" and "M3"). In its turn, each one of these submodes contain another child submode, which is activated by the "SHIFT" button.

With this setup, the profile will behave in the following way:

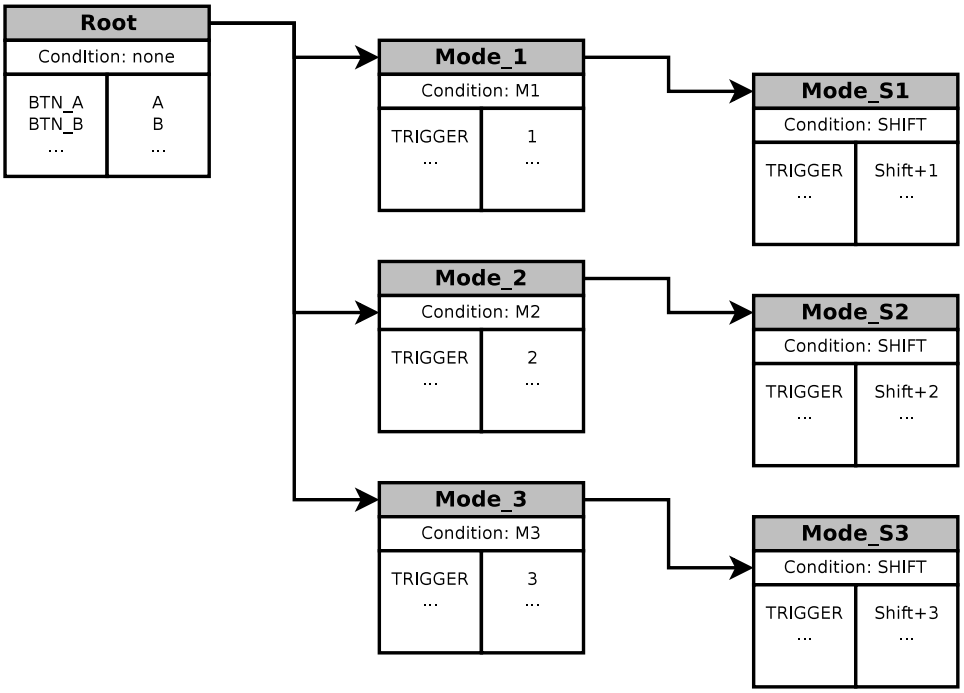

Figure 6.2: Modes in cascade

- For "A" or "B" buttons, the driver will always emit the corresponding action mapped by *root* mode, as it's not overriden by any of the submodes.
- For "TRIGGER" button, the driver will initially choose the appropiate submode by checking the state of the "M1", "M2" and "M3" buttons. Then, it will check "SHIFT" in order to decide if the *shifted* child mode must be used: if so, then the action from the shifted submode will be emitted, else the one from the submode.

6.2 Driver API

As seen on previous chapters, the *jsmapperdev* kernel module creates a *jsmapXX* device node under */dev/input/* for every gaming-type device attached to the sysrem. These device nodes are the entry point for the userspace API, which is used to load profiles into the driver, this is, mapping actions to device controls.

A program wanting to interact with the device would open the device node (using regular *open()* system calls), then issue the required *ioctl()* calls

to load the whole mapping set into the driver.

These are some of the IOCTL control code supported by the API:

- **JMIOCGVERSION**: returns API version
- **JMIOCGNAME**: returns associated device name
- **JMIOCGBUTTONS**, **JMIOCGAXES**: returns number of buttons / axes in device
- **JMIOCCLEAR**: clears device mapping
- **JMIOCSBUTTONACTION**: assigns an action to the given button
- **JMIOCSAXISACTION**: assigns an action to a band of the given axis
- **JMIOCADDMODE**: adds a new mode

The whole set of IOCTL codes supported by the API, along with the structures to be passed as parameters, are defined in the *jsmapper_api.h* header file.

6.3 Profiles

Mapping profiles are XML files containing the definition of the mapping to be to load into the device. They can be create using a regular text editor, and contains the whole list of actions, modes, mappings, etc... that can be atomically loaded into the device.

The user is expected to create a profile for every game it/she want to play with the device, every profile contaning the mapping from device controls to game actions, by means of simulated keyboard and mouse actions.

Check annex A “User manual” for complete profile format documentation.

6.3.1 Device Maps

Device maps are special XML files that provide mapping between joystick elements “natural· names and their related kernel numerical ID. They are needed because there is no easy way to obtain the ”actual“ name of an

element, other than for the very basic X & Y axes and some specific buttons ("FIRE", i.e.).

Device maps thus, provide a way to make writing a profile an easier task by providing easy-to-remember names for the elements, so instead of referring to them by a raw number a more "natural" name can be used.

A bunch of device map files are shipped with the project, and JSMapper userspace tools will automatically try to use them, by finding the appropiate one on a device name basis.

An special CLI tool (*jsmapper-device*) is included to make writing *device map* files an easier task.

6.3.2 Profile loading

Profiles can be loaded into the device by using the provided *jsmapper-ctrl* tool, which is the userspace CLI-based tool for JSMapper.

Example 6.1 shows how to load a profile into the device:

Listing 6.1: Loading a profile

```
$ jsmapper-ctrl -l profile.xml
```

The same tool can be used also to clear the state of the driver, so it stops filtering and restores normal device behaviour, as displayed in example 6.2:

Listing 6.2: Clearing device

```
$ jsmapper-ctrl -c
```

Chapter 7

Planning & Economical Analysis

7.1 Planning

The project development phase was divided in "milestones", each one trying to deliver a sort of "functional" product (even if in a very bare state). These were the milestones defined:

- *0.0 Initial proof-of-concept*: initial research, including writing a minimal input event handler kernel module capable of filtering device source events and subsitute them by "fake" ones.
- *0.1 Basic kernel module*: in this phase, the very bare kernel module created in previous milestone was refactored, in order to actually be able of mapping arbitrary buttons on the device to simple keyboard key press & release events. Initial programming API was also included here.
- *0.2 User space tools*: started implementation of the C/C++ warapper library, plus the userspace binaries to manipulate the kernel module and load profiles into it.
- *0.3 Mode support*: added support for mode hierarchy, both in kernel module and in C/C++ library.
- *0.4 Axis mapping*: added support for axis mapping through bands.
- *0.5 Macros*: added support for "macro" actions.

- *0.6 Mouse events*: added support for mouse actions, including simulating mouse movement.
- *1.0 Initial release*: final polishing, including testing, bugfixing, and writing wiki-based user manual.

Table 7.1 shows the project scheduling, according to the main milestones defined, and their associated Gantt diagram (dates shown are aproximative).

The whole development of the project has entirely been made only over spare time, in order to render it compatible with my current full-time job.

7.2 Economical analysis

The whole cost analysis is detailed in table 7.2.

The project has been developed using exclusively FOSS (Free and Open Source Software) tools, thus it didn't require spending any money on software licenses. Because of that, the cost of this project relies solely on the "hardware" part (computers and testing devices used), plus the cost of the development time, assuming a reasonable cost per hour.

The resulting software product is licensed using the GPLv2 license, and final users will be able to make use of it completely free of charge.

Figure 7.1: Gantt diagram

Hardware		Purpose		Cost
	Main PC (custom)	main development		900,00 €
	Netbook Toshiba NB-301	occasional development		350,00 €
	Saitek X-45	testing		120,00 €
	Logitech Wingnan Extreme	testing		40,00 €
	Total			**1.410,00 €**

Software		Version	License	Cost
	Ubuntu	(various)	GPL	0,00 €
	openSUSE	(various)	GPL	0,00 €
	Fedora	18	GPL	0,00 €
	QtCreator	2.5	GPL	0,00 €
	Kile	2.1	GPL	0,00 €
	Dia	0.97	GPL	0,00 €
	Calligra Suite	2.5.5	GPL	0,00 €
	FlightGear	2.8	GPL	0,00 €
	Total			**0,00 €**

Development		Hours	Cost / Hour	Cost
	0.0 - Initial research	14,0	50,00 €	700,00 €
	0.1 - Basic kernel module	10,0	50,00 €	500,00 €
	0.2 - User space tools	22,5	50,00 €	1.125,00 €
	0.3 - Mode support	13,0	50,00 €	650,00 €
	0.4 - Axis mapping	8,5	50,00 €	425,00 €
	0.5 - Macros	24,0	50,00 €	1.200,00 €
	0.6 - Mouse events	25,0	50,00 €	1.250,00 €
	1.0 Initial release (incl. doc)	60,0	50,00 €	3.000,00 €
	Total			**8.850,00 €**

Total				
	Hardware			1.410,00 €
	Software			0,00 €
	Development			8.850,00 €
	Total			**10.260,00 €**

Figure 7.2: Cost analysis

Chapter 8

Conclusions

8.1 Achievements

The original motivation behind this project was to bring to Linux OS some of the advanced features offered by joystick manufacturers for the Windows operating system, by means of the propietary drivers shipped along wth the product. In addition, the idea was to come with a very generic solution, not tighted to any particular device, so a large number of Linux users could benefit from it.

I think that this objective has been succesfully achieved, as the solution offered satisfies all the conditions above. Moreover, the way the project is structured makes easy to add new features, complementary tools, etc... in the future, so there should be no need for large rewrites of the project codebase.

A secondary, more personal objective was to get started with the basics of Linux kernel development, which I also consider to be a succesfully achieved objective: even if this project has dealt only with a very specific portion of the Linux kernel codebase (the input subsystem), it has allowed me to learn about how kernel modules are constructed, how they interact with the system and communicate with the outside, and how to deal at kernel level with low level programming aspects such as memory allocations, IRQ contexts, threads and work queues.

8.2 Future

8.2.1 Improvements

Even as the basic objectives for the project has been succesfully achieved, there's still room for a lot of improvements and / or possible future developments:

- *GUI applet*: this would increase the usability of the project, as final users would not need to deal with the command line in order to load game profiles into the driver. The GUI frontend could be something as simple as an applet-type application displaying an icon on the desktop status bar, through which the user would select the profile to be loaded using a drop-down menu. Or else a more sophisticated application which would allow to, i.e., associate game profiles to game executables, the monitorize the system to automatically load the appropiate profile when the game executable is started.
- *GUI profile builder*: this would be a large GUI application, which would allow the user to interactively create a game profile by letting it enter keystrokes, associate actions with device controls, create modes, etc..., all from a confortable GUI. This would also dramatically improve the user-friendliness of the project, as the user would be abple to create a game profile without manually editing an XML file.
- *Extending actions*: so far, the user can map either keystrokes or mouse movements to any device control. It might also be interesting to be able to define complex, mixed sequences of keyboard and mouse actions to be mapped, including a better fine-grain control of how the sequence of executed, the timing of every step involved, etc...
- *Generic-device mapping*: this means extending the mapping capabilities of JSMapper to any input device, not only joysticks and similar. This i.e. to arbitrary assign keystrokes, macros, etc... to mouse buttons or to an specific key on the keyboard. Although this would need only relatively few changes to the kernel module implementation, it would represent a certain drift from the original project objectives (which were centered around gaming).

The two first, GUI-related items could be implemented without any further modification to the current implementation, just by using the provided API library to deal with the driver. This means, that in fact, they could be completely independent projects, not related at all to the JSMapper project itself.

Moreover, in order to provide a satisfactory desktop integration, it would probably be better if desktop-specific applets are implemented, so i.e. both KDE and GNOME desktops feature its own GUI applet.

For the GUI profile builder, a generic cross-platform toolkit such as Qt could be used.

The other two suggested items would require more extensive changes in the existing code base, specially on the kernel module.

8.2.2 Kernel inclusion

Another interesting way to explore for the project is the possibility of the inclusion of *jsmapperdev* module into the mainline kernel sources, so the module gets automatically included with every Linux distribution, just as current *joydev* module is.

This might of course imply some potentially extensive modifications on the existing code base, in order to match the very strict requirements of the kernel maintainers regarding code inclusion. Despite this, I keep the inclusion into mainline kernel as a mid-term objective.

Glossary

Action In JSMapper, a "fake" event (or events) to be simulated, such as key press & release events, or mouse movement events, whenever a control on the device is operated.

API Acronym for "Application Program Interface".

Axis In a game device, a control providing a continuous range of discrete values, usually designed to represent magnitudes which are analogous in "real world", such as direction on a wheel drive, throttle, brake, etc... .

Band In JSMapper, an interval of values defined over an axis, which can be used to map actions to specific portions of an axis range.

Button In a game device, a control featuring two possible states, "pressed" and "released".

CLI Acronym for "Command Line Interface".

CMake A popular cross-platform, open-source build system developed by Kitware, Inc.

Condition In JSMapper, the circumstances defining if a given mode is active or not, such an specific button in "pressed" state or else an axis positioned on a given band.

Control In an input or game device, each of the elements present on it designed to provide user interaction.

D-BUS An open-source inter-process communications system for Linux originally developed by RedHat, Inc.

Event In Linux input susbsystem, a notification sent by a hardware device whenever the user interacts with any of the controls on it.

FOSS Acronym for “Free and Open Source Software”.

Game Device An input device specifically designed to improve user experience with computer games, such as a joystick or a wheel drive.

Git A distributed source control software extensively used in FOSS projects, originally developed by Linus Torvalds.

Gnome Desktop environment and graphical user interface for Linux, focused on accessibility and user friendliness.

GUI Acronym for “Graphical User Interface”.

HAL Acronym for “Hardware Abstraction Layer”. In Linux, a service providing an abstract view of the hardware attached to the system, plus notifications to userspace about device pluging and unplugging.

Hat Button In a game device, usually a “hat”-type button providing direction (up/down/left/right) control. They are usually implemented internally using either regular buttons or axes.

HID Acronym for “Human Interface Device”, a device class defined by USB specification specifically designed for user interaction devices, such as a keyboard, a mouse, etc... Most input and game devices lays within this class..

HOTAS Acronym for “Hands-On-Throttle-And-Stick”, usually a device composed of both a joystick and a throttle, featuring a number of buttons, so a plane pilot can perform most usual tasks without taking hands out from the controls.

Input Device A pluggable computer device intended for user input, such as a keyboard, a mouse, a graphical tablet, etc....

Input Subsystem In Linux, the part of the kernel responsible of managing attached input devices and providing notifications about user interaction up to userspace.

IOCTL Acronym for “Input/Output Control”, a generic mechanism for device-specific input/output operations which cannot be expressed by regular system calls, such as device parametrization.

IRQ Acronym for “Interrupt Request”, a hardware mechanism through which a device claims attention from kernel, i.e. to notify user interaction .

IRQ Context Calling context in which a function is called from within an ISR, and where some restrictions apply.

ISR Acronym for "Interrupt Service Routine", the part of the kernel responible of handling IRQ requests.

KDE Desktop environment and graphical user interface for Linux, focused on customizability and targeted for advanced users.

Kernel The main component of most computer operating systems, usually running under *protected* (or *privileged*) mode, and responsible of managing system resources among running applications..

Kernel Module Loadable module for an operating system kernel, usually providing extended capabilities for that kernel, such as support for specific hardware devices.

Linux Unix-like computer operating system, assembled under the model of free and open source software development and distribution.

Macro In JSMapper, an action defining an arbitrary sequence of keys to be sent.

Mode In JSMapper, a set of actions mapped to controls available on a device, plus a "condition" defining under which circumstances the mode is active..

OS Acronym for "Operating System".

Profile In JSMapper, a set of modes and conditions for a device targeting an specific game or program, and which can be serialized to disk for easy loading into the device.

PS/2 A device port found in old IBM Personal System/2 computers.

Serial In computing, an physical interface to which information transmits one bit at a time. In PC computers, it was available usually throught an RS-232 port.

Serialization In JSMapper, the act of storing a profile definition into an XML file, so it can be restored later.

Switch Button In a game device, a control featuring different fixed, available positions, each of one internally implemented using a regular button control.

TAR A file format widely used in Unix world, containing a file tree layout and usually stored on disk or tape, either in compressed or uncompressed form.

Thread In Linux kernel, an independent flow of control that operates within the same address space as the kernel, and usually used to perform background actions.

USB Acronym for “Universal Serial Bus”.

Userspace Memory area and processor mode in which user applications run under modern operating systems.

Windows Propietary graphical interface and operating system, developed, marketed, and sold by Microsoft, Inc.

Wine Acronym for “Wine Is Not an Emulator”, a compatibility layer capable of running Windows applications on several POSIX-compliant operating systems, such as Linux.

Workqueue In Linux kernel, a low-level mechanism which allows an arbitrary operation to be ”queued“ onto a service queue that gets dispatched by a background thread.

X11 (or “X Window System”), a computer software and network protocol providing the basis for the Graphical User Interface on modern Unix and Linux systems.

XML Acronym for “eXtensible Markup Language”.

Bibliography

[1] *KernelHacking for Newbies*
`http://kernelnewbies.org/KernelHacking`

[2] *Linux Input Drivers*
Vojtech Pavlik
`http://atrey.karlin.mff.cuni.cz/~vojtech/input/`

[3] *The Linux USB Input Subsystem, Part I*
Brad Hards (LinuxJournal - 2003)
`http://www.linuxjournal.com/article/6396`

[4] *The Linux USB Input Subsystem, Part II*
Brad Hards (LinuxJournal - 2003)
`http://www.linuxjournal.com/article/6429`

[5] *Linux Device Drivers, 2nd Edition* (Online Book)
Alessandro Rubini & Jonathan Corbet (O'Reilly - June 2001)
`http://www.xml.com/ldd/chapter/book/`

[6] *Linux Loadable Kernel Module HOWTO*
Bryan Henderson (The Linux Documentation Project - 2006)
`http://www.tldp.org/HOWTO/Module-HOWTO/index.html`

[7] *Kernel APIs, Part 2: Deferrable functions, kernel tasklets, and work queues*
M. Tim. Jones (IBM developerWorks - 2010)
`http://www.ibm.com/developerworks/linux/library/l-tasklets/index.html`

[8] *The New Work Queue Interface in the 2.6 Kernel*
Robert Love (LinuxJournal - 2003).
`http://www.linuxjournal.com/article/6916`

[9] *CMake Documentation*
`http://www.cmake.org/cmake/help/documentation.html`

[10] *Google Test Framework Documentation*
`http://code.google.com/p/googletest/wiki/Documentation`

[11] *LaTeX WikiBooks*
`http://en.wikibooks.org/wiki/LaTeX`

Appendix A

User manual

A.1 Overview

Once the kernel module has been installed and loaded, it should attach itself to any game plugged into the system. To verify this, just check for the creation of a "/dev/input/jsmapX" node after device connection.

Using JSMapper for the end user should be a pretty straightforward process:

- First of all, you might need to create a *device map* for your device. This is needed only if none of the provided ones adequately matches your device.
- Then, you must write a *game profile*, this is, an XML file describing the game actions and how they are mapped to the device controls.
- Finally, you must *loading* the profile into the device

A.2 Device Maps

Device map files provide mapping between "natural" element names (such as "FIRE", "TRIGGER", "X", etc...) and the numeric ID of the device item inside Linux kernel. This is just provided so, when writing a game profile, instead of using a confusing bunch of numerical IDs such as 0, 1, etc... to refer to buttons and axes, we can use a more easier to remember element names, thus driving to a more readable game profile format.

Listing A.1 displays a device map file, specifically, the one for Saitek X-45 device.

Listing A.1: Saitek X-45 Device Map file

```
<?xml version="1.0"?>
<device name="Saitek X45 Flight Control Stick">
    <button id="0" name="TRIGGER" />
    <button id="1" name="A" />
    <button id="2" name="B" />
    <button id="3" name="FIRE" />
    <button id="4" name="D" />
    <button id="5" name="MOUSE_CLICK" />
    <button id="6" name="SHIFT" />
    <button id="7" name="C" />

    <button id="8" name="MODE_1" />
    <button id="9" name="MODE_2" />
    <button id="10" name="MODE_3" />

    <button id="11" name="AUX_1" />
    <button id="12" name="AUX_2" />
    <button id="13" name="AUX_3" />

    <button id="14" name="HAT1_UP" />
    <button id="15" name="HAT1_RIGHT" />
    <button id="16" name="HAT1_DOWN" />
    <button id="17" name="HAT1_LEFT" />

    <button id="18" name="HAT3_UP" />
    <button id="19" name="HAT3_RIGHT" />
    <button id="20" name="HAT3_DOWN" />
    <button id="21" name="HAT3_LEFT" />

    <button id="22" name="MOUSE_UP" />
    <button id="23" name="MOUSE_RIGHT" />
    <button id="24" name="MOUSE_DOWN" />
    <button id="25" name="MOUSE_LEFT" />

    <axis id="0" name="X" />
    <axis id="1" name="Y" />
    <axis id="2" name="ROTARY_1" />
    <axis id="3" name="RUDDER" />
    <axis id="4" name="THROTTLE" />
    <axis id="5" name="ROTARY_2" />
    <axis id="6" name="HAT2_X" />
    <axis id="7" name="HAT2_Y" />
</device>
```

As seen above, device map format is pretty straightforward: just a bunch of "button" and "axis" elements, each one defining both the internal, numerical, element ID and their assigned name.

The device map also should feature a "name" attribute on the root element: this name SHOULD match the internal name of the device provided by Linux kernel: right now, the device map to use must be specified when

loading a profile, but in a future this probably will be done automatically based one device name.

A.2.1 Creating a device map

This step is needed only if none of the predefined device map files under "/usr/share/jsmapper/devices" is suitable for your device. Listing A.2 show how to create a new device map file using the provided *jsmapper-device* tool:

Listing A.2: Creating a new device map file

```
$ jsmapper-device -c <path-to-device-map.xml>
```

This will query the attached device to determine its name, number of buttons and axes, and will generate a bare device map file on the given output path. The program will try to determine an appropiate name for the elements found, for those it can't it will leave them with their numerical IDs.

Once the initial device map is created, you can see how element names are mapped by invoking again the program, this time in "live view" mode. Listing A.3 shows how to do it:

Listing A.3: Checking a device map file

```
$ jsmapper-device -v <path-to-device-map.xml>
```

This will display all the device elements, their mapped names, and their current vaues. By operating device elements you can check the names assigned to them, then open the device map into an editor and change the mapped names as you detect which one is every element. The view will be refreshed automatically as you edit the device map or change device values. Once the device map for your device is done, you can start creating the game profile by using the element names in there.

A.2.2 Why device map files are needed?

Linux kernel input system features a number of meaningful constants to identify both buttons (BTN_xxx) and axes on a device (ABS_xxx). Unfortunately, device drivers (usually the generic HID driver, in fact) don't do a very good job at providing proper ID values for the elements detected on the device: either they report an unappropiate ID (such as ABS_RUDDER for an axis which is clearly not the rudder...), a complete meaningless one (such as BTN_TRIGGER_HAPPY5, i.e.), or even a numeric value not mapped

to any defined constant. So, having to rely on element names based on these identifiers was clearly not an option, as they would be hard for the end user to figure out the right element name in every case.

Another possibility was to simply use a numeric index, corresponding to the position or the element inside the order in which they are reported by the device (0..numButtons - 1 for buttons, i.e.): however, having to rely on numeric values for every element wouldn't make the profile writing task easier that with kernel IDs.

Thus the needed for a custom, XML file based mapping between device elements and "names", so the user is free to give every button and axis a meaningful name, then use it inside the profile to map actions to it.

A.3 Creating a Profile

A.3.1 Game profiles

A game profile is simply an XML file containing a list of *actions*, which can be simple key presses, mouse movements, complex macros, etc..., and a list of *mode* definitions, which map actions against available buttons and axis on the device.

The basic structure of a game profile file is as follows:

- a root *"profile"* node. It might include a *"name"* attribute, just for reference.
- an *"actions"* sub-node, containing a list of *"action"* elements. These are described in the Actions section.
- a single *"mode"* sub-node, containing the root mapping between device elements and actions. Sections A.3.3 "Buttons" and A.3.3 "Axes" describe how these mappings are defined.

Listing A.4 displays an example of a very simple game profile, which maps the ENTER key to the FIRE button on the device:

Listing A.4: Example profile

```
<?xml version="1.0"?>
<profile name="Test">
    <description>A very simple profile </description>
    <actions>
        <action name="Intro" type="key" key="ENTER" >
            <description>
                An simple action which presses ENTER key
            </description>
        </action>
    </actions>
    <mode name="Root">
        <description>The root mode</description>
        <button id="FIRE" action="Intro" />
    </mode>
</profile>
```

The *"description"* elements are optional, and merely for informative purposes (they are not loaded into the device).

A.3.2 Creating actions

Actions are defined by *"action"* items inside the *"actions"* element of the root *"profile"* element, with the following attributes:

- a mandatory *"name"* attribute, which is later used when mapping the action to either a button or an axis band.
- a mandatory *"type"* attribute, which specified the type of action (see below).
- an optional *"filter"* attribute, which specifies if original device source event is filtered out after launching the action, or else is allowed to continue its way to target application (default is true).
- an optional *"description"* sub-element, containing a textual description of the action, which is only for informational purposes.

Currently, there are 5 types of actions supported:

- *keystroke* actions
- *button* actions
- *axis* actions
- *macro* actions
- *null* actions

Keystroke actions

Keystroke actions allows user to simulate a keystroke (either a single key, or a shortcut-like composed using a single key plus optional modifiers such as CTRL, ALT,...) being pressed on the keyboard. A keystroke action definition should include the following elements:

- a *"type"* attribute with *"key"* value element
- a mandatory *"key"* attribute, specifying the key to send (see section A.3.5 "Element names" for supported key names)
- an optional *"modifiers"* attribute, defining the possible key modifiers to use such as SHIFT, CTRL, ... (to specify multiple modifiers, use ";" between them)
- an optional *"single"* attribute, defining if the action should hold down the key as long as the button is pressed or else release it immediately (default is false)

Example Listing A.5 displays an example of some keystroke actions:

Listing A.5: Sample keystroke actions

```
<actions>
    <action name="Intro" type="key"
                    key="ENTER" />
    <action name="Spc" type="key" single="true"
                    key="SPACE" />
    <action name="FileOpen" type=key␣single="true"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣key="O"␣modifiers="LEFTCTRL;LEFTALT"␣/>
</actions>
```

How they work Keyboard-based actions behaves differently depending on the value of the (optional) "single" attribute:

- if *single="false"* (default), then a "key press" event for the key (and, previously, for the optional modifiers, if any) is sent whenever the target button is pressed (if the action is mapped to a button), or else the axis value enters the target "band" (see section A.3.3). Then, a "key release" will be sent when the button is released or axis band exited. This will result on the key being self-repeating as long as the button is held down, i.e., just like if a regular key on your keyboard os held down.
- if *single="true"*, then a single pair key press and release events will be sent whenever the button is pressed down or axis band entered, and no action will be taken on button release or axis band exit.

Button actions

Button actions are used to simulate mouse button events. In fact, they are fairly similar to keystroke actions, only difference being the XML tag used to define the action. A button action definition should include the following elements:

- a *"type"* attribute with *"button"* value element
- a mandatory *"button"* attribute, specifying the button to send (see section A.3.5 "Element names" for supported button names)
- an optional *"modifiers"* attribute, defining a possible key modifiers such as CTRL and SHIFT to use (to specify multiple modifiers, use ";" between them)
- an optional *"single"* attribute, defining if the action should hold the button down, or else release it immediately (thus generating an immediate click)

Example Listing A.6 displays an example of a button action:

Listing A.6: Sample button action

```
<actions>
    <action name="MouseLeft" type="button" key="LEFT" />
</actions>
```

Axis actions

Axis actions are used to simulate axis movement events. Right now, only relative axes are supported (which are the ones used by mouse, i.e.). An axis action definition should include the following elements:

- a *"type"* attribute with *"axis"* value element
- a mandatory *"axis"* attribute, specifying the button to send (see section A.3.5 "Element names" for supported axis names)
- an optional *"step"* attribute, specifying the number of axes units used by every step (default is 1)
- an optional *"single"* attribute, defining if the axis should keep moving as long as the button is pressed (default is true)
- an optional *"spacing"* attribute, defining how many milliseconds to wait between steps (default is 100 ms)

Example Listing A.7 displays an example of some possible axis actions:

Listing A.7: Sample axis actions

```
<actions>
    <action name="MouseLeft" type="axis"
                    axis="X" step="-1" spacing="5" />
    <action name="MouseRight" type="axis"
                    axis="X" step="1" spacing="5" />
    <action name="MouseUp" type="axis"
                    axis="Y" step="-1" spacing="5" />
    <action name="MouseDown" type="axis"
                    axis="Y" step="1" spacing="5" />
</actions>
```

How they work For axis actions for which *"single"* attribute is false, a driver thread will be generating axis movement events for as long as the source button is held down. Every event will contain the number of steps defined in the action, and they will be spaced by the specified amount of time in ms.

For single-type axis actions, only one movement step will be generated.

Macro actions

Macros are simply a fixed, time-spaced sequence of keys to be sent that can get assigned to either a button or an axis band. They are intended to be used to automate a series of steps by sending a bunch of key events to the game.

In order to define a "macro" type action, the following elements should be included:

- a *"type"* attribute with *"macro"* value
- a *"keys"* element, containing itself the key sequence to launch, each one as a *"key"* element inside it.
- an optional *"spacing"* attribute, defining the number of milliseconds to wait between generated key presses (default is 250 ms).

Example Listing A.8 displays an example of a macro action which attempts to open an hypothetical "File" menu (which could be opened using *ALT-F* shortcut) featuring a "Print" command, accessible through the "P" shortcut once the menu is opened:

Listing A.8: Sample macro action

```
<actions>
    <action name="Macro" type="macro" spacing="100" >
        <description>Invokes File menu, Print command</description>
        <keys>
            <key key="F" modifiers="LEFTALT" />
            <key key="P" />
        </keys>
    </action>
</actions>
```

How they work Macros are launched whenever the target element is activated, meaning by that the button is pressed (for a button), or the axis enters the assigned band range (in case of an axis band). This means that, for every key defined, a couple of key press and release events will be generated, then the driver will wait for the specified amount of time before continuing with the next key defined in the sequence.

It's important to note that macros are NOT cancellable (by releasing the button before they have finished, i.e.), nor they self-repeat automatically (i.e. by holding down the button.): every time the button is pressed, a single macro sequence will be launched.

Null actions

Null actions are an advanced feature, mostly used to "hide" device events to the target application by means of assigning them an empty action that does nothing, but filters out the original source event.

Null actions are specified by using *"none"* value for the *"type"* attribute in action definition.

A.3.3 Mapping actions

Actions are mapped to device buttons and axes by including the appropiate clause into a *"mode"* declaration, as explained in the sections below.

Buttons

In order to map an action to a button, a *"button"* node is included inside a mode declaration. The node must include the following elements:

- an *"id"* attribute specifying the name of the button this mapping applies to, as defined in the target device's map file.
- an *"action"* attribute, specifying the name of the action to launch whenever the button is operated. The given name should match one of the action's name define on the *"actions"* part of the profile.

Example Listing A.9 displays an example of a game profile mapping some keyboard actions to buttons on the device:

Listing A.9: Mapping buttons

```
<?xml version="1.0"?>
<profile name="Test">
    <description>A very simple profile </description>
    <actions>
        <action name="Intro" type="key"
                    key="ENTER" />
        <action name="Spc" type="key"
                    single="true" key="SPACE" />
        <action name="FileOpen" type=key"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣single="true"␣key="O"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣modifiers="LEFTCTRL;LEFTALT"␣/>
␣␣␣␣␣␣␣␣</actions>
␣␣␣␣<mode␣name="Root">
␣␣␣␣␣␣␣␣<description>The␣root␣mode</description>
␣␣␣␣␣␣␣␣<button␣id="TRIGGER"␣action="Intro"␣/>
␣␣␣␣␣␣␣␣<button␣id="FIRE"␣action="Spc"␣/>
␣␣␣␣␣␣␣␣<button␣id="A"␣action="FileOpen"␣/>
␣␣␣␣</mode>
</profile>
```

Axes

Actions can also be mapped to axis present in the device, by defining *"bands"* on them, then assigning different actions to each one of the bands created.

Axis mappings are defined by including an *"axis"* element inside a mode declaration, and including the following elements:

- an *"id"* attribute specifying the name of the button this mapping applies to, as defined in the target device's map file.
- a number of children *"band"* elements, each one featuring an *"action"* attribute specifying an action, plus a *"low"* and *"high"* value attributes defining the axis band value range.

How they work Defining band ranges on an axis is equivalent to defining "virtual" buttons on it, so actions can be mapped to them in a similar way to that of normal buttons on the device.

This is:

- moving axis position into the band is equivalent to "pressing" the button
- moving axis position outside the band, is thus, equivalent to "releasing" the button

This means that, i.e., a keystroke action is mapped to an axis band, the assigned key will get "pressed" when axis position is moved into this band, then will self-repeat as long as the axis is left there, and wil get "released" when axis exits the band.

It's also possible to leave some value bands unassigned. In such a case, no action will be taken whenever axis position is inside the band.

Example Listing A.10 displays an example of a game profile mapping some actions to device axes:

Listing A.10: Mapping axes

```
<?xml version="1.0"?>
<profile name="Test">
    <description>
        A simple profile mapping cursor keys to X-45's␣HAT2␣axes
␣␣␣␣</description>
␣␣␣␣<actions>
␣␣␣␣␣␣␣␣<action␣name="Up"␣type="key"␣key="UP"␣/>
```

```
␣␣␣␣␣␣␣␣<action␣name="Down"␣type="key"␣key="DOWN"␣/>
␣␣␣␣␣␣␣␣<action␣name="Left"␣type="key"␣key="LEFT"␣/>
␣␣␣␣␣␣␣␣<action␣name="Right"␣type="key"␣key="RIGHT"␣/>
␣␣␣␣</actions>
␣␣␣␣␣␣␣␣<mode␣name="Root">
␣␣␣␣␣␣␣␣<description>The␣root␣mode</description>
␣␣␣␣␣␣␣␣<axis␣id="HAT2_X">
␣␣␣␣␣␣␣␣␣␣␣␣<band␣low="-1"␣high="-1"␣action="Left"␣/>
␣␣␣␣␣␣␣␣␣␣␣␣<band␣low="1"␣high="1"␣action="Right"␣/>
␣␣␣␣␣␣␣␣␣␣␣␣</axis>
␣␣␣␣␣␣␣␣<axis␣id="HAT2_Y">
␣␣␣␣␣␣␣␣␣␣␣␣<band␣low="-1"␣high="-1"␣action="Down"␣/>
␣␣␣␣␣␣␣␣␣␣␣␣<band␣low="1"␣high="1"␣action="Up"␣/>
␣␣␣␣␣␣␣␣</axis>
␣␣␣␣</mode>
</profile>
```

In this example, four actions are defined to represent cursor keys, then mapped to bands on *HAT2_X* and *HAT2_Y* axes, which are the two axes a Saitek X-45 device returns for the second HAT-style button it features on top of the stick (the other one, by contrary, is represented by 4 regular buttons). Both axes offer only 3 possible values (-1, 0, 1), being "0" the value returned in repose mode. With the mapping above, the HAT2 will behave as a regular pad, moving cursor along the four directions.

A.3.4 Modes

A *"mode"* is simply a mapping between joystick controls and the action to perform for them when invoked. As seen in the past examples, every profile has one, and only one, *root mode*, containing the base assignments for the profile.

In order to add new modes, they must be created as *children* of the root mode, and must also feature an *activation condition* that will determine when the submode is active.

Example Listing A.11 displays an example of a game profile mapping some actions to device axes:

Listing A.11: Modes

```
<?xml version="1.0"?>
<profile name="Test">
    <description>
        A simple testing profile
    </description>
    <actions>
        <action name="Intro" type="key" key="ENTER" />
        <action name="Spc" type="key" key="SPACE" />
    </actions>
    <mode name="Root">
        <description>The root mode</description>
        <button id="FIRE" action="Intro" />
        <mode name="Shift">
            <condition type="button" id="SHIFT"/>
            <button id="FIRE" action="Spc" />
        </mode>
    </mode>
</profile>
```

In this example, two simple actions are created, one triggering an ENTER key (*"Intro"*) and the other one the SPACE key (*"Spc"*). In root mode, FIRE button is mapped to *"Intro"* action. Then, a child mode is created, which gets activated using the SHIFT button: in it, FIRE button is overriden, by assigning it the *"Spc"* action instead of *"Intro"*.

How they work As seen on the example, new modes get defined simply by adding a nested *"mode"* element inside another one, usually under the root one (which is mandatory). Contrary to root mode, however, any child mode must feature a *"condition"* element, which describes the activation condition for such a mode.

Right now, only button-type condition is supported, which means that the mode will be active if the given button (by mean of the *"id"* attribute) is held down. Section *A.3.4 "Mode activation"* below for more information about how mode selection works at runtime.

A *child* mode can either add new mappings to the profile, or either override parent's one with its own assignments. Any element not assigned by a child mode will simply inherit the assignment made in its parent mode.

Mode activation At runtime, whenever any button or axis on the device is operated, the module tries to determine which is the correct mode to apply by checking their associated activation conditions and the current state of the device.

In order to do so, it starts from the root mode, and checks if any of its children modes is active by checking its activation conditions. If so, then it repeats the process recursively, starting from the child mode.

Listing A.12 displays an example of a multi-mode profile, featuring a root mode and 3 submodes, every one of which features also a "shift state" submode on its own:

Listing A.12: Modes and submodes

```
<?xml version="1.0"?>
<profile target="Saitek␣X45␣Flight␣Control␣Stick"
                 name="Test">
    <description>A simple testing profile </description>
    <actions>
        <action name="A" type="key" key="A" />
        <action name="B" type="key" key="B" />
        <action name="1" type="key" key="1" />
        <action name="2" type="key" key="2" />
        <action name="3" type="key" key="3" />
        <action name="Shift_1" type="key" key="1"
                        modifiers="LEFTSHIFT" />
        <action name="Shift_2" type="key" key="2"
                        modifiers="LEFTSHIFT" />
        <action name="Shift_3" type="key" key="3"
                        modifiers="LEFTSHIFT" />
```

```
    </actions>
    <mode name="Root">
        <description>The root mode</description>
            <button id="A" action="A" />
            <button id="B" action="B" />
            <mode name="Mode_1">
                <condition type="button" id="MODE_1"/>
                <button id="TRIGGER" action="1" />
                    <mode name="Mode_1_Shift">
                        <condition type="button" id="SHIFT"/>
                        <button id="TRIGGER" action="Shift_1" />
                    </mode>
            </mode>
            <mode name="Mode_2">
                <condition type="button" id="MODE_2"/>
                <button id="TRIGGER" action="2" />
                <mode name="Mode_2_Shift">
                    <condition type="button" id="SHIFT"/>
                    <button id="TRIGGER" action="Shift_2" />
                </mode>
            </mode>
            <mode name="Mode_3">
                <condition type="button" id="MODE_3"/>
                <button id="TRIGGER" action="3" />
                <mode name="Mode_3_Shift">
                    <condition type="button" id="SHIFT"/>
                    <button id="TRIGGER" action="Shift_3" />
                </mode>
            </mode>
    </mode>
</profile>
```

As seen above, the root mode contains 3 submodes, each of one is activated through the corresponding positions of the *MODE* switch on the device (which gets translated to 3 separate buttons, internally). Each of these, additionally, features a submode, which gets activated through the *SHIFT* button, which overrides the assignments made in parent mode.

The last part if is the most interesting part of this example, as it shows how a single activation button (*SHIFT*) can be used for 3 different submodes, as mode selection algorithm will consider it only after having reached first the corresponding parent mode, depending on *MODE* switch button state.

A.3.5 Element names

As JSMapper mapping works at kernel level, keys, buttons, etc... are referred by the identifiers used at kernel level, specifically the *KEY_xxx*, *BTN_xxx*, *REL_xxx* constants defined in *<linux/input.h>* file (without the prefix). In order to make profile writing easier, the whole list of supported valid item names can be obtained using *jsmapper-ctrl* tool.

Example A.13 shows how to use *jsmapper-ctrl* to display the list of supported target key names:

Listing A.13: Obtaining target key names

```
$ jsmapper-ctrl --keys
```

Example A.14 shows how to use *jsmapper-ctrl* to display the list of supported target button names:

Listing A.14: Obtaining target button names

```
$ jsmapper-ctrl --buttons
```

Example A.15 shows how to use *jsmapper-ctrl* to display the list of supported target axis names:

Listing A.15: Obtaining target axis names

```
$ jsmapper-ctrl --axes
```

Determining key symbols When using keyboard layouts other than the standard "US" layout, getting to know which is the right symbol for a given key can be a little tricky, as the "real" hardware key identifier might not match which is printed on it.

The right symbol to use can be determined by using the "evtest" input helper tool, like seen in example A.16:

Listing A.16: Getting key names

```
$ evtest /dev/input/event0
...
Event: time 1349944227.260230, type 1 (Key), code 43 (BackSlash), value 0
...
```

This will dump keys pressed on the terminal: the symbol name to use is the one enclosed between parenthesis right after "code" value ("BackSlash", in this case). The symbol displayed MUST be converted to upper case, as *JSMapper* uses uppercase names for keys (so, "BACKSLASH" is the right name to use in the profile).

Note: the right /dev/input/eventX *device to use can be different among systems. If there is more than one "eventX" element present, then try each one until you get some output when pressing keyboard keys.*

A.4 Using profiles

Once the game profile is ready, it can be loaded into the kernel module by using *jsmapper-ctrl*, which is the userspace tool for jsmapper module. A full list of the options it offers by calling it with the *–help* parameter, as seen in example A.17:

Listing A.17: Getting help

```
$ jsmapper-ctrl --help
```

A.4.1 Loading a profile

In order to load a profile, user must call *jsmapper-ctrl* with the *–load* (or *-l* for short) to specify the profile file to load. Also, the device map file to use must also be specified, as seen in example A.18:

Listing A.18: Loading a profile

```
$ jsmapper-ctrl -l <profile-file> -d <device-map-file>
```

This is needed because the device map file specifies how button and axes names get translated to device identifiers.

A.4.2 Clearing device

Once finished, device can be cleared by using *-c* option, as seen in example A.19:

Listing A.19: Clearing device

```
$ jsmapper-ctrl -c
```